Proyecto Fin de Máster en Ingeniería de Computadores.

Técnicas de simplificación de la política de reemplazamiento cache Probabilistic Escape LIFO

Autor:
Enrique Sedano Algarabel

Directores:
Daniel Ángel Chaver Martínez
Fernando Castro Rodríguez

Máster en Investigación en Informática. Facultad de Informática.
Universidad Complutense de Madrid.
Curso 2009-2010.

Autorización

El abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: Técnicas de simplificación de la política de reemplazamiento cache Probabilistic Escape LIFO, realizado durante el curso académico 2009-2010 bajo la dirección de Daniel Ángel Chaver Martínez y Fernando Castro Rodríguez en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Enrique Sedano Algarabel

A mi familia y amigos

Proyecto Fin de Máster en Ingeniería de Computadores.

Técnicas de simplificación de la política de reemplazamiento cache Probabilistic Escape LIFO

Autor:
Enrique Sedano Algarabel

Directores:
Daniel Ángel Chaver Martínez
Fernando Castro Rodríguez

Máster en Investigación en Informática. Facultad de Informática.
Universidad Complutense de Madrid.
Curso 2009-2010.

La frase más excitante que se puede oir en la Ciencia, la que sin duda ha llevado a más descubrimientos, no es “Eureka!” sino “Anda, qué curioso!”

Isaac Asimov

Índice general

Índice de figuras

Índice de tablas

Resumen

La memoria cache es el mecanismo más extendido a la hora de salvar la gran diferencia de prestaciones entre el procesador y la memoria principal. Una política de reemplazamiento que utilice adecuadamente la información del comportamiento de los bloques durante su estancia en la cache podrá escoger correctamente los bloques a eliminar de la cache, reduciendo así la tasa de fallos. La política Probabilistic Escape LIFO, presentada recientemente, se basa en la observación experimental de que el número de reusos de los bloques en cache es, en general, mayor que uno, pero muy inferior al valor de asociatividad de la cache. Combinando las ventajas de los algoritmos de inserción dinámica y de la política LRU, la política Probabilistic Escape LIFO obtiene buenos resultados, aunque a costa de una implementación hardware muy costosa.

Este trabajo presenta una serie de técnicas de simplificación de esa política que reducen la cantidad de recursos necesarios para su implementación sin incurrir por ello en un deterioro excesivo de sus prestaciones, mejorando la relación entre complejidad hardware y calidad del algoritmo.

Abstract

Cache memory is the most extended mechanism for saving the increasing gap between processor and memory performance. A replacement policy that uses correctly the information regarding the behaviour of the blocks while they are in the cache will be able to choose correctly the blocks to evict, thus reducing the miss rate. The Probabilistic Escape LIFO policy, recently presented, is based on the experimental observation that the number of reuses of the blocks in the cache is usually higher than one, but much lower than the associativity of the cache. Combining the advantages of dynamic insertion policies and LRU algorythm, the Probabilistic Escape LIFO policy obtains good results at the expense of a very complex hardware implementation.

This work presents several techniques of simplification for that policy that reduce the amount of resources needed for its implementation with a negligible performance degradation, increasing with that the ratio between hardware complexity and algorithmical quality.

Palabras clave

- Cache
- Políticas de reemplazamiento
- Simplificación
- Probabilistic Escape LIFO
- Memoria
- Profiling
- Grano grueso
- Rendimiento
- LRU

Keywords

- Cache
- Replacement policies
- Simplification
- Probabilistic Escape LIFO
- Memory
- Profiling
- Coarse grain
- Performance
- LRU

Capítulo 1

Introducción

La enorme diferencia de rendimiento entre procesador y memoria no hace sino aumentar con cada nueva arquitectura que aparece en el mercado. Mientras que la capacidad de cómputo sigue creciendo imparable según la Ley de Moore, la velocidad a la que la memoria es capaz de suministrar y almacenar datos crece mucho más despacio [1] (Figura 1.1). La aparición de las arquitecturas multiprocesador no ha hecho sino agravar este problema, convirtiéndolo en uno de los principales cuellos de botella de la computación [2].

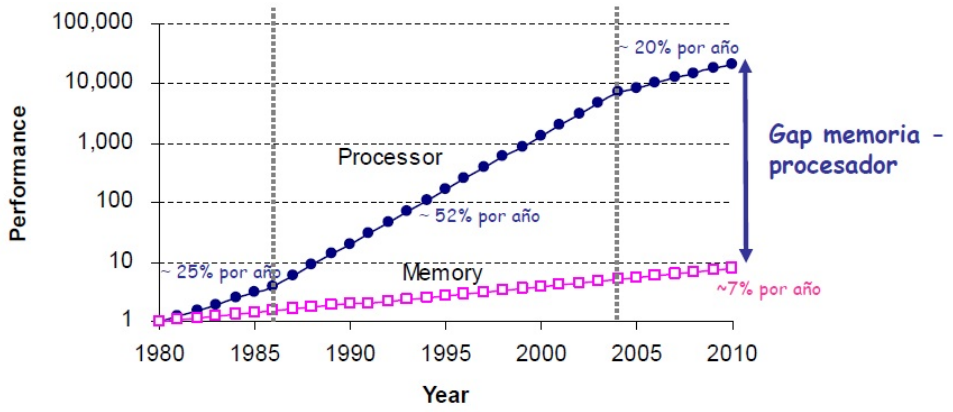

Figura 1.1: Incremento del gap memoria-procesador en el tiempo

Durante los últimos años, los investigadores han tratado de resolver el problema de la memoria desde perspectivas muy variadas. Así, se han propuesto diversas estrategias a nivel de procesador. En esta familia encontramos la predicción de dependencias entre loads y stores [3], la predicción de valores [4] o la predicción de la dirección [5], entre otras. En el otro extremo, encontramos técnicas a nivel de la DRAM de sistemas multi-core orientadas a planificar los accesos a eśta de forma que el rendimiento global del sistema mejore [6]. Sin embargo, el campo en el que más se ha investigado para lograr solventar el problema de la memoria ha sido en la jerarquía de cache situada entre el procesador y la memoria principal.

La cache es una memoria de tamaño pequeño en comparación con la memoria principal del sistema, pero varios órdenes de magnitud más rápida. A medida que el espacio en el chip ha ido siendo cada vez mayor, este tipo de memoria se ha ido jerarquizando, incluyendo varios niveles cada vez de mayor capacidad pero, consecuentemente, más lentos que el anterior. Paulatinamente, el espacio dedicado a la jerarquía de memoria cache ha ido ocupando una porción más importante del silicio disponible. Actualmente, casi tres cuartas partes del área de los chips están dedicadas a este componente (Figura 1.2).

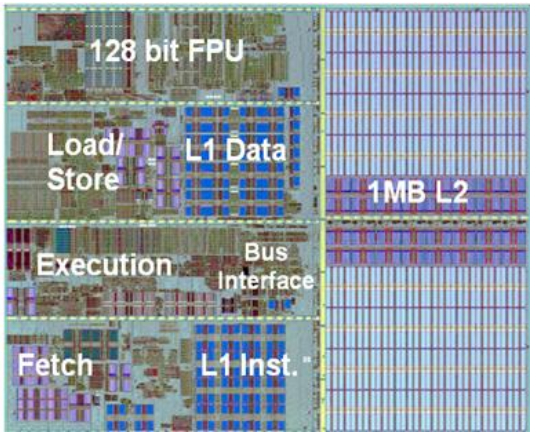

Figura 1.2: Layout del procesador AMD LLano (Febrero 2010)

Existe una amplia gama de propuestas en la literatura que tratan de mejorar la gestión de la cache y realizar un aprovechamiento óptimo de los recursos que hay a nuestra disposición. Así, los mecanismos de prefetching y caches no bloqueantes [7] tratan de solapar los fallos de cache mediante la ejecución de instrucciones independientes de los mismos. Una de las formas más extendidas de mejorar el rendimiento de una cache asociativa por conjuntos es la aplicación de políticas eficientes de reemplazamiento de bloques.

Como demostró Belady [8], el bloque de un conjunto que más conviene reemplazar es el que va a ser referenciado dentro de más tiempo. Las distintas políticas de reemplazamiento cache intentarán predecir cual será ese bloque a partir de la información pasada para de este modo conseguir provocar el mínimo número posible de fallos de cache. Una de las políticas más comunes es la clásica LRU: Implementada como una cola de últimos accesos, cuando un bloque es referenciado se coloca en la cabeza de la cola (posición Most Recently Used -MRU-), y si se ha de eliminar un bloque, se escoge el ubicado al final de la cola (posición Least Recently Used -LRU-). Esta política está diseñada bajo la asunción de que, por localidad temporal, si un bloque se ha usado recientemente tiene muchas probabilidades de ser usado de nuevo próximamente. Aunque este esquema ofrece un buen comportamiento en bastantes aplicaciones, adolece de varios problemas. En primer lugar, si un bloque que acaba de entrar en la cache no va a volver a ser usado hasta dentro de mucho tiempo, o no va a ser usado en absoluto, una política óptima lo seleccionaría como candidato para ser reemplazado de inmediato. Sin embargo, la política LRU lo mantiene en la cache durante todo el tiempo que tarda en llegar hasta la última posición de la cola, ocupando por tanto una vía que podría ser empleada por un bloque con mayor frecuencia de accesos. Además, si la aplicación trabaja de forma iterativa con un conjunto de N datos y una cache

de tamaño M, con N > M, es muy probable que la política genere N fallos en cada iteración.

Muchas son las políticas que se han propuesto [9–15] para tratar de resolver los problemas previamente descritos, así como algunos otros. De entre todas estas estrategias de reemplazamiento cache, nuestro trabajo extiende una propuesta muy reciente llamada Probabilistic Escape LIFO [9] (peLIFO), que según la información ofrecida por sus autores y la obtenida en nuestras comprobaciones, mejora sustancialmente el rendimiento con respecto a la política LRU y a otras propuestas recientes, tanto single-core como multi-core. Los autores observaron que la cantidad de reúsos a corto plazo de muchos bloques de cache es mayor que uno para una gran variedad de aplicaciones, por lo que no es conveniente eliminar de la cache un bloque recién llegado. Sin embargo, el número de reúsos a corto plazo es siempre inferior a la asociatividad de la cache, por lo que en ocasiones la política LRU mantiene algunos bloques en cache mucho más tiempo del necesario. La política peLIFO, en cambio, trata de ajustar dinámicamente el número de vías necesarias para mantener los bloques con reúsos a corto plazo y destina el resto a bloques que serán reutilizados a largo plazo.

Este trabajo presenta una serie de técnicas que buscan reducir considerablemente la complejidad algorítmica y el coste hardware de la implementación de la política peLIFO sin que por ello la calidad de los resultados del algoritmo se vean excesivamente deteriorados. Nuestro objetivo, por tanto, es mejorar tanto como sea posible la relación entre complejidad hardware y calidad del mecanismo propuesto en [9]. La política peLIFO establece periódicamente tres posiciones de su lista de bloques residentes en la cache a partir de las cuales poder decidir qué bloques pueden ser descartados. En cada reemplazamiento se decide, de manera dinámica, cuál de esos tres puntos se elegirá como posición

desde la que empezar a buscar un bloque que cumpla los requisitos para ser eliminado.

Nuestro trabajo presenta dos aproximaciones distintas al problema de simplificación de este algoritmo. La primera de ellas consiste en utilizar profiling para dar valores fijos a las posiciones desde las que buscar un candidato para ser reemplazado, manteniendo que el algoritmo decida en cada reemplazamiento cuál de los tres puntos dados es el adecuado. Por su parte, la segunda propuesta, principal de este trabajo, introduce un sistema de decisión de grano grueso para determinar dinámicamente el punto a partir del cual se buscarán los bloques a reemplazar, empleando dicha decisión durante periodos largos de tiempo, en lugar de reevaluarla en cada reemplazamiento como la política original. Mediante la aplicación de estas dos políticas hemos logrado una importante reducción del overhead hardware que supone la implementación de la política peLIFO mientras que conseguimos que el deterioro de los resultados se mantenga en niveles muy bajos. Así, la diferencia media del porcentaje de tasa de fallos entre nuestras propuestas y peLIFO se mantiene en torno al 0.5 %.

En el capítulo 2 de este documento ofrecemos una visión de conjunto de los antecedentes y el estado del arte de las políticas de reemplazamiento en cache. El capítulo 3, núcleo de este trabajo, presenta y desarrolla las propuestas que planteamos. El entorno de simulación en que las hemos probado se explica en el capítulo 4. En el capítulo 5 mostramos los resultados experimentales obtenidos para finalmente exponer nuestras conclusiones y apuntar a futuras vías de investigación en el capítulo 6.

Capítulo 2

Trabajo relacionado

La literatura referente a políticas de reemplazamiento es muy extensa. Este capítulo recoge algunas de las propuestas más interesantes y representativas para entornos tanto single-core como multi-core. En último lugar, detallamos el funcionamiento de la política Probabilistic Escape LIFO, la cual este proyecto utiliza como punto de partida.

2.1. Políticas de reemplazamiento single-core

Las políticas de reemplazamiento cache han experimentado un gran desarrollo a lo largo de los años, a medida que se ha ido demostrando su importancia. Partiendo de las propuestas más básicas, como LRU (Least Recently Used) [16], LFU (Least Frequently Used), FIFO (First In First Out), Round-Robin o reemplazamiento aleatorio [1] se han ido desarrollando diversas familias de políticas orientadas a obtener un mejor aprovechamiento de la cache a través de un reemplazamiento inteligente.

Un primer grupo de trabajos tratan de obtener implementaciones eficientes

para la política LRU. De este modo, encontramos la implementación Pseudo-LRU, también conocida como Tree-LRU. La idea de este algoritmo es considerar un árbol binario de búsqueda para los elementos de la cache. Cada nodo del árbol indica la dirección a seguir para encontrar el siguiente elemento a reemplazar. El árbol se recorre guiado por dichos flags, y a la hora de actualizar con un dato D, se recorre el árbol hasta dicho valor D y se modifican las direcciones de todos los nodos que se han ido recorriendo para que apunten en el sentido contrario. Esta política fue utilizada en las arquitecturas 486 de Intel, así como en algunos miembros de la familia PowerPC. Más tarde, [17] propone una generalización de la política anterior para el uso de árboles con un nivel de ramificación superior a dos. Otra propuesta de este tipo es la que se muestra en [18], donde los árboles se mantienen como binarios, pero algunos de los nodos intermedios no sólo almacenan la dirección actual a tomar, sino un pequeño historial de datos previos, y utiliza toda esa información a la hora de decidir el camino a tomar.

Otra línea de investigación de gran relevancia es la de la predicción de Dead Blocks. Un Dead Block se define como aquel bloque que está en la cache pero no va a volver a ser referenciado antes de que sea reemplazado. Un diseño inteligente sería aquél que a la hora de elegir un bloque a reemplazar escoja un Dead Block en lugar de uno que vaya a ser referenciado otra vez más adelante (denominados Live Blocks). La predicción de Dead Blocks se ha realizado, además, desde enfoques diversos. Por un lado, se ha tratado de predecirlos mediante un análisis de la dirección de los datos [19] para, a través de la historia pasada, predecir lo que sucederá en el futuro de la ejecución. También existen aproximaciones software, como [20], donde se propone ayudar a la cache mediante la inserción de etiquetas que marcan bloques que deben mantenerse y que pueden eliminarse, o [21] donde se propone que sea el compilador el que

detecte, mediante el planteamiento de diversos algoritmos, los Dead Blocks. Por supuesto, existen políticas puramente hardware como las que encontramos en [22,23]; el primero de ellos con las vistas puestas en el consumo, buscando los bloques que no se utilizarán en el futuro para poder desactivarlos, mientras que el segundo emplea predicción basada en contadores que se incrementan según distintos criterios (número de referencias al bloque o tiempo de permanencia en la cache), y al llegar a un umbral dado, pasan a considerarse muertos y por tanto disponibles para el reemplazo. Una última política perteneciente a esta famila es la reciente [24]. En este trabajo se define el concepto de cache burst (o ráfaga) como el conjunto de accesos contiguos que un bloque recibe cuando se encuentra en la posición MRU de su set sin intervención de referencias a ningún otro bloque del mismo set. A partir de esa idea se propone un predictor basado no en accesos sino en bursts. Por un lado, se propone la idea de llevar la cuenta de ráfagas en lugar de la de accesos para determinar si el bloque ha pasado a estar muerto o no. Por otro se expone el uso de trazas de bursts para llevar a cabo este tipo de predicción. Los resultados experimentales demuestran que el rendimiento es mucho mejor que la predicción basada en conteo de referencias.

Otra aproximación distinta la encontramos en [10]. Los autores se fijan en el hecho de que cuando varios fallos ocurren a la vez, los ciclos de espera se amortizan entre todos, por lo que son menos dañinos que los fallos aislados. Teniendo esto en cuenta, proponen una política de reemplazamiento (a la que llaman LIN) que trata de mantener en cache aquellos bloques sobre los que un fallo ocurriría de forma aislada. Si bien esta política obtiene buenos resultados, la principal aportación de este trabajo es el Set Sampling: Los autores proponen una política de reemplazamiento híbrida integrada por dos, LRU y LIN. En cada reemplazamiento se decide cuál usar en base a una evaluación dinámica del rendimiento de cada una de ellas (figura 2.1). Para realizar dicha evaluación,

se realiza una competición entre un pequeño grupo de conjuntos que sólo usan LRU y otro pequeño grupo que sólo usa LIN. El grupo ganador (es decir, aquel en el que se producen menos fallos) fija la política para el resto de la cache.

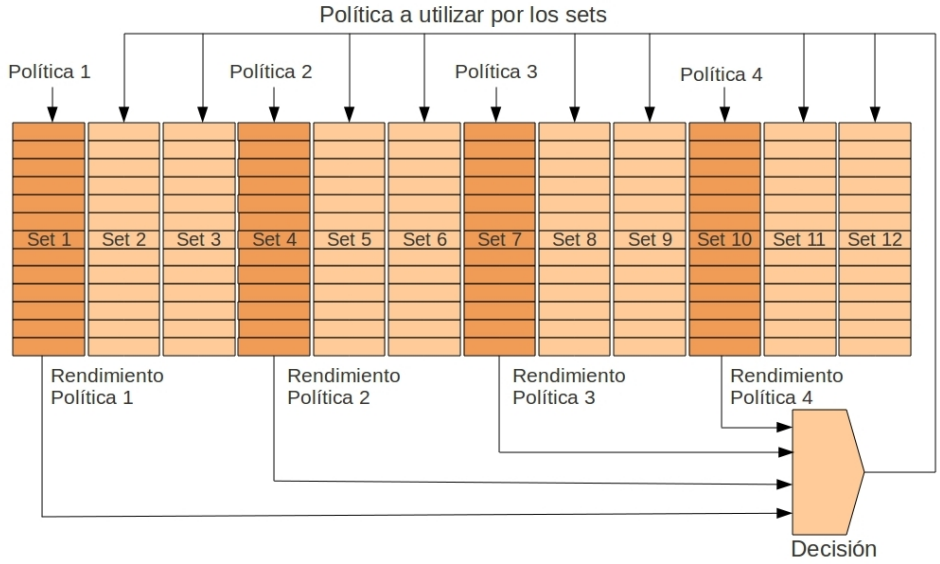

Figura 2.1: Esquema de cache con mecanismo de Set Sampling

Otro trabajo de los últimos años [11] busca suavizar el problema que ocurre con la política LRU cuando el conjunto de datos de trabajo es mayor que el tamaño de la cache, tratando de conservar al menos parte de éste. Los autores presentan dos conjuntos de políticas: estáticas y dinámicas. Para el primer grupo proponen una sencilla política similar a LRU (llamada LIP), en la que el bloque expulsado es también el que se referenció hace más tiempo (el LRU), pero en la que un bloque no es insertado como el MRU sino como el LRU. Así, si el bloque recién insertado no se vuelve a referenciar pronto probablemente será expulsado, y se conseguirán mantener en cache bloques con reúso a largo plazo. Basándose en LIP, los autores proponen también otra política estática: BIP, en la que sólo una parte de los bloques nuevos se insertan como LRU y el resto como MRU.

Como propuesta dinámica, los autores presentan la denominada política de inserción dinámica, o DIP, una política híbrida que aplicando Set Sampling combina BIP y LRU. Para lograr una implementación eficiente de esta política los sets de la cache se distribuyen ente un número determinado de regiones de tamaño uniforme. Un set de cada una de la regiones estará destinado a cada una de las dos políticas entre las que decidir (BIP y LRU), y el resto se establecerán como seguidores. De este modo los autores logran reducir de manera considerable la cantidad de bits requeridos para identificar cada uno de los componentes de la política.

Complementariamente, y de manera auxiliar a todas las políticas que acabamos de ver encontramos dos estrategias para minimizar la penalización por fallo. La primera de ellas es la Victim Cache [13], consistente en la inclusión de una cache de pequeño tamaño y totalmente asociativa a la que se envían los bloques que salen de la cache principal durante un reemplazo. De este modo, si un bloque es referenciado poco después de haber sido eliminado de la cache, es muy probable que todavía pueda ser encontrado en la Victim Cache, ahorrando por tanto el coste de acceso a la memoria principal. En segundo lugar, en un artículo muy reciente [12] se propone utilizar aquellos conjuntos de la cache que están siendo poco utilizados para albergar bloques de conjuntos altamente sobrecargados. En primer lugar estudian una opción estática (SSBC), en la que cada conjunto está asociado con su conjunto más alejado. Cuando un bloque es reemplazado, se mira si su conjunto asociado está infrautilizado, y en caso afirmativo se alberga en él. En segundo lugar estudian una propuesta dinámica (DSBC), en la que se mantiene una lista de los conjuntos menos saturados que se va actualizando en tiempo de ejecución. Cuando se produce un reemplazamiento, el bloque eliminado se emplaza en el conjunto menos saturado de la lista. La propuesta es similar a la Victim Cache, aunque más eficiente, pues

no requiere una cache totalmente asociativa adicional.

2.2. Políticas de reemplazamiento multi-core

Con la proliferación de los sistemas multi/many-core en los últimos años, han surgido políticas que tratan de adaptarse eficientemente a estos entornos. El principal desafío es conseguir que la política funcione bien en los niveles compartidos de la jerarquía, en los que al estar accediendo varios threads se produce interferencia destructiva. Aunque en un principio en las caches de niveles compartidos se aplicaban directamente políticas originalmente diseñadas para entornos monoprocesador, como LRU (existen estudios que demuestran que en ocasiones la aplicación directa de una política clásica es suficiente), en muchos casos la interferencia lleva a grandes pérdidas de rendimiento, haciéndose evidente la necesidad de diseñar técnicas que contemplen información sobre dicha interferencia para tomar sus decisiones.

Así, en [14], los autores proponen repartir periódicamente las vías de un nivel compartido de cache entre los threads que lo comparten, teniendo en cuenta las necesidades de cada uno de ellos. Para ello, en el momento de repartir las vías se estima, por medio de unos contadores, cuánto se beneficiará cada thread (es decir, cuánto reducirá el número de fallos que va a cometer) si recibe x vías (para x desde 2 hasta asociatividad) respecto a si recibe sólo 1 (la propuesta exige que al menos se asigne 1 vía a cada thread). El reparto elegido será aquel que minimice el número total de fallos.

Otra propuesta reciente es [25]. En ella, los autores proponen la política TADIP, híbrida entre [11] y [14]. A la hora de decidir la posición en que se insertarán una serie de bloques en la cache, la política debe escoger entre

insertarlos según la política LRU o BIP. As, se encuentra ante un árbol de decisiones binarias de tamaño N (siendo N la cantidad de bloques a insertar), y por tanto, con un espacio de búsqueda de tamaño 2N. Para buscar la mejor de las combinaciones de decisiones, la que lleve a un uso más óptimo de la cache, el artículo propone tres estrategias: (1) Uso de profiling, con el que determinar estáticamente la mejor cadena de decisiones, aunque apuntan que cuando la cantidad de bloques a insertar es alta, el espacio de búsqueda se hace demasiado extenso. (2) El empleo, en caso de que N sea pequeña, de Set Sampling para comparar mediante fuerza bruta todas las alternativas y quedarse con la mejor. (3) La propuesta principal del artículo, que parte de la observación de que no todas las decisiones son dependientes las unas de las otras, y por tanto algunas de ellas pueden optimizarse de forma independiente.

Finalmente, en [26] se propone otra técnica para reducir la interferencia en el nivel compartido de cache. Periódicamente, cada conjunto de la cache se asigna dinámicamente a un único thread (el que se estima que más lo está usando), de tal modo que sólo el thread propietario de un conjunto es capaz de emplazar/reemplazar bloques en/de el mismo. Además, existe una pequeña zona adicional de la cache en la que se sitúan aquellos bloques a los que no se les permite acceder al conjunto al que pertenecerían en condiciones normales.

2.3. La política Probabilistic Escape LIFO

Mención aparte merece [9], artículo en el que se propone la política de reemplazamiento Probabilistic Escape LIFO (peLIFO), en la cual se fundamenta nuestro trabajo. Los autores de este artículo motivan su propuesta basándose en la observación de que aunque las políticas de inserción dinámica (vistas en el apartado 2.1) funcionan bien con conjuntos grandes de bloques de un

solo uso, en multitud de aplicaciones hay una gran cantidad de bloques cuyo número de reusos a corto plazo es mayor que uno, por lo que en estos casos las políticas de inserción dinámica toman la decisión errónea de sustituir algunos bloques demasiado pronto. Sin embargo, la cantidad de reusos de un bloque es, en general, menor al valor de la asociatividad de la cache, por lo que una política de tipo LRU estaría manteniendo los bloques durante más tiempo del necesario. La política peLIFO busca ofrecer una solución intermedia, aprendiendo dinámicamente cual es el número de vías que son necesarias para satisfacer los reusos a corto plazo y reservando el resto para almacenar bloques que produzcan aciertos en usos a largo plazo.

Como punto de partida para la política peLIFO, los autores toman la política de reemplazamiento LIFO, en la que el primer bloque en entrar es el primero en salir, y proponen optimizaciones que mejoran su precario rendimiento. Para lograr dicho objetivo, la familia de políticas pseudo-LIFO utilizan, además de la tradicional Pila de últimos accesos que usa LRU, la llamada *Fill Stack* o pila de llenado. Dicha pila es de tamaño igual a la asociatividad de la cache, y en la cima (posición cero) de la misma siempre encontraremos el bloque que más recientemente se ha insertado, como se muestra en la figura 2.2. La idea básica consiste en reemplazar bloques de la parte superior de la *Fill Stack*, dejando la parte baja de la pila sin tocar, de modo que pueda satisfacer accesos a largo plazo.

La política está construida alrededor del concepto de Probabilidad de Escape. Ésta se entiende como la probabilidad de que un bloque vuelva a ser referenciado en el futuro. Dicho de otra forma, podemos decir que la Probabilidad de Escape para una posición k de la *Fill Stack* es la probabilidad de que los bloques de cache tengan aciertos en posiciones de la *Fill Stack* superiores a k. Dicha probabilidad se calcula como el número de bloques de cache que han

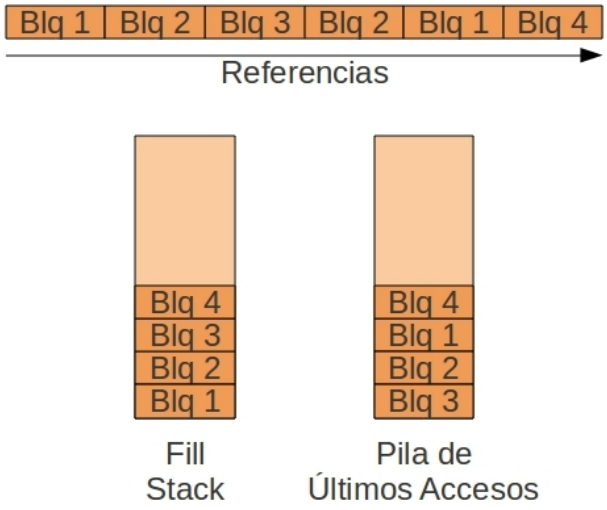

Figura 2.2: Orden de llenado de *Fill Stack* y pila de últimos accesos

tenido al menos un acierto a partir de la posición k dividido entre el número total de bloques con que se ha rellenado la cache.

En la política LIFO convencional, a la hora de escoger un bloque a reemplazar, siempre extraeremos el que se encuentra en la cima de la pila (figura 2.3 (a)). En peLIFO, en lugar de eso, se definen los llamados *Puntos de Escape* (en [9] se definen tres): Se trata de posiciones de la *Fill Stack* donde la Probabilidad de Escape disminuye de manera notable, por lo que a partir de ellos no merece la pena mantener los bloques en cache y se escogerán como posiciones para realizar los reemplazamientos (figura 2.3 (b)). El modo de proceder en un reemplazamiento es el siguiente: si se estima dinámicamente (más adelante explicaremos cómo) que peLIFO no está funcionando bien, se utiliza la política LRU. Por el contrario, si peLIFO está funcionando adecuadamente se selecciona uno de los tres *Puntos de Escape* (P_i). A continuación la política selecciona como bloque a reemplazar el bloque más cercano a la cabeza de la *Fill Stack* que cumpla (1) que no ha tenido acierto en la posición en la que se encuentra

actualmente y (2) que ocupe una posición mayor o igual que P_i. De este modo se estarán realizando los reemplazamientos en la posición supuestamente más adecuada de la zona alta de la cola, y conservando las zonas bajas para reúsos a largo plazo.

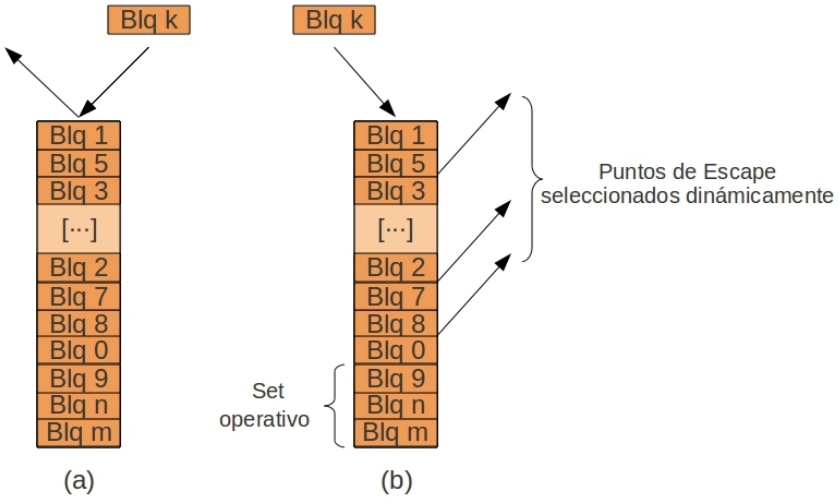

Figura 2.3: Reemplazamiento en las políticas LIFO y peLIFO

Para llevar a cabo el cálculo de las *Probabilidades de Escape*, los autores de la propuesta emplean un array de contadores saturados denominado epCounter. Este array tiene una cantidad A de contadores igual a la asociatividad de la cache. Cuando se produce un acierto en un bloque alojado en la posición k de la *Fill Stack*, todos los contadores desde la última posición en que dicho bloque produjo un acierto hasta k-1 se incrementan en una unidad, y la última posición en que el bloque produjo un acierto se actualiza a k.

Con esa información, el algoritmo calculará periódicamente las *Probabilidades de Escape*. Éste se efectúa en tres pasos: (1) Primero, cada valor del epCounter se redondea a la siguiente potencia de dos. (2) A continuación, cada valor positivo se sustituye por su logaritmo en base dos, y (3) finalmente, se resta cada valor a A y se vuelve a almacenar en el epCounter.

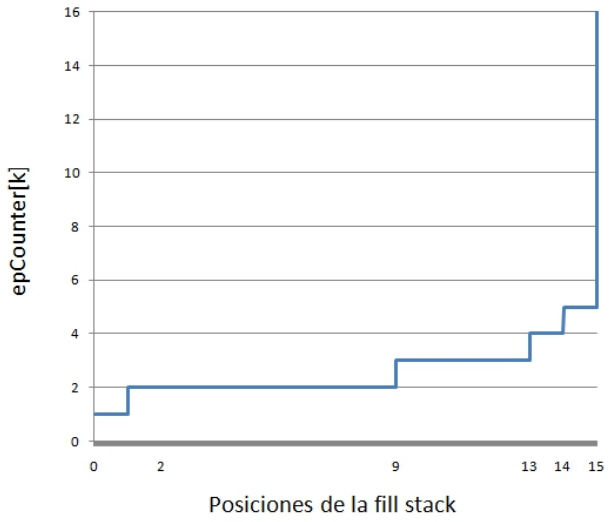

Figura 2.4: Forma de epCounter con respecto a la posición k de la *Fill Stack*

Una vez realizado este proceso, los valores almacenados en epCounter son valores monotonamente crecientes con valores de 0 a A. Si se observa una gráfica con los resultados que se obtienen de este cálculo, como la mostrada en la figura 2.4, se pueden observar varios puntos, o codos, donde el valor se incrementa (y por lo tanto la Probabilidad de Escape disminuye). La política peLIFO escoge los tres codos más cercanos a la cima de la pila como los *Puntos de Escape*.

A la hora de tomar la decisión de qué política usar para efectuar el reemplazamiento (peLIFO con cada uno de los tres *Puntos de Escape* o LRU) se utiliza Set Sampling. El sistema, como se ha explicado en la sección 2.1, reserva algunos de los conjuntos de la cache para evaluar las políticas entre las que escoger, y el resto de sets emplearán la política que mejores resultados haya obtenido.

Dado que el comportamiento de una aplicación varía a lo largo de su tiempo de ejecución, es necesario que los valores de los *Puntos de Escape* se vayan

recalculando cada cierto tiempo. Con este fin se definen los conceptos de época y fase. Se entiende una época como el tiempo que transcurre entre dos cálculos sucesivos de las *Probabilidades de Escape*, el cual se lleva a cabo cada N inserciones de bloques nuevos en la cache. Dicho valor N será siempre una potencia de dos. Los valores obtenidos en el cálculo de las nuevas *Probabilidades de Escape* se comparan con los obtenidos al final de la época anterior. Cuando la diferencia entre ambos valores en alguna posición por encima del tercer *Punto de Escape* es superior a un umbral dado, se dice que se produce un cambio de fase. Podemos definir entonces una fase como el tiempo durante el que las *Probabilidades de Escape* de las posiciones de cabeza de la *Fill Stack* no experimentan un cambio brusco de una época a la siguiente.

Dentro de una fase, la primera época siempre se ejecutará únicamente en modo LRU, con el objetivo de limpiar la cache de valores de la fase anterior que pudiesen contaminar los resultados de la actual. Una vez terminada la época LRU, se realiza el cálculo de los tres *Puntos de Escape*, y el resto de épocas de la fase se llevan a cabo empleando el algoritmo descrito en los párrafos anteriores (figura 2.5).

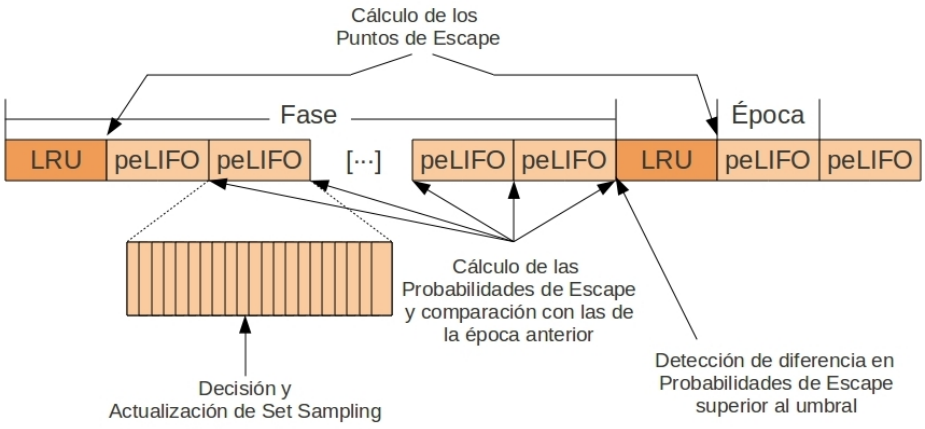

Figura 2.5: Esquema de funcionamiento de la política peLIFO

A costa de una importante complejidad algorítimica, la política peLIFO demuestra obtener unos resultados muy superiores a los del resto de políticas contra las que se compara. Los autores muestran el rendimiento de su algoritmo frente a Dead Block Prediction convencional, DIP y Victim Cache. Para aplicaciones single-core la política muestra una mejora media del 10.3 % [9] con respecto a la política LRU (figura 2.6), obteniendo importantes mejoras de rendimiento en seis de los catorce benchmarks que se utilizan de prueba. A la hora de probar la política en un entorno multi-core los resultados demuestran que peLIFO también consigue mejor rendimiento que diversas políticas planteadas especialmente para este tipo de entornos a pesar de que peLIFO no considera información específica de threads de ejecución. De media, para entornos multi-core, la política discutida en este apartado obtiene una mejora media del CPI de un 19.4 % frente al obtenido con LRU (figura 2.7) [9].

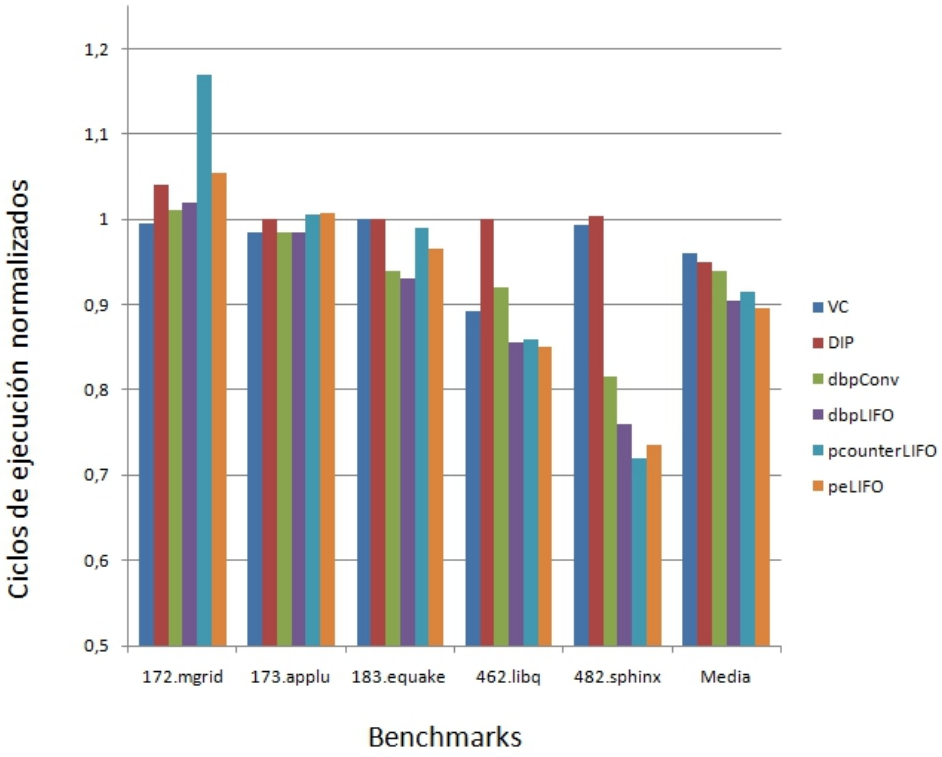

Figura 2.6: Resultados para peLIFO en entorno single-core

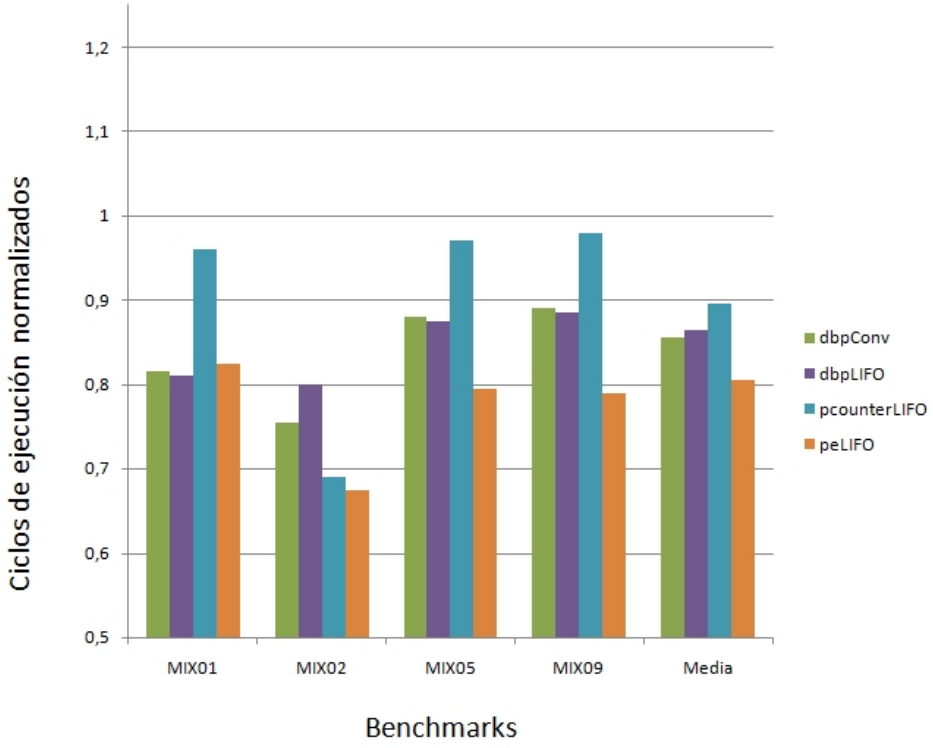

Figura 2.7: Resultados para peLIFO en entorno multi-core

Capítulo 3

Estrategias propuestas

Como hemos comentado, la política peLIFO obtiene unos resultados muy buenos, casi siempre superiores a los existentes hasta la fecha. Sin embargo, esta mejora se consigue a cambio de un incremento sustancial de la complejidad hardware del mecanismo. Dado que el coste en recursos es muy alto, el objetivo principal de este trabajo consiste en proponer simplificaciones en el diseño que ofrezcan una mejora significativa en el compromiso entre coste y rendimiento, logrando una reducción drástica de la complejidad del mecanismo a cambio de sacrificar algo del rendimiento del mismo.

3.1. Uso de *Puntos de Escape* fijos

Si realizamos un análisis del uso de los distintos *Puntos de Escape* a lo largo de la ejecución de las aplicaciones, podemos comprobar que un muy elevado porcentaje de las selecciones recaen sobre un conjunto muy reducido de *Puntos de Escape*. Atendiendo a la información proporcionada por el autor en [9], en 13 de las 14 aplicaciones evaluadas las selecciones se concentraban más del 74 %

de las veces en tres de los posibles *Puntos de Escape*. Dicha concentración ascendía por encima del 90 % de las selecciones en 8 de los benchmarks (tabla 3.1).

Benchmark	***Puntos de Escape* más usados**	**Total**
171.swim	2 (26 %), 1 (26 %), 15 (22 %)	74 %
172.mgrid	15 (63 %), 13 (7 %), 14 (6 %)	76 %
173.applu	2 (31 %), 1 (28 %), 15 (23 %)	82 %
179.art	2 (84 %), 6 (9 %), 0 (2 %)	95 %
181.mcf	0 (59 %), 1 (31 %), 15 (3 %)	93 %
183.equake	1 (94 %), 15 (2 %), 5 (2 %)	98 %
401.bzip2	4 (51 %), 7 (25 %), 15 (2 %)	78 %
403.gcc	11 (22 %), 15 (13 %), 10 (13 %)	48 %
429.mcf	1 (78 %), 5 (12 %), 2 (4 %)	94 %
433.milc	1 (58 %), 15 (20 %), 2 (8 %)	86 %
462.libq	1 (95 %), 15 (3 %), 2 (<2 %)	>99 %
470.lbm	2 (87 %), 3 (5 %), 1 (5 %)	97 %

Tabla 3.1: Porcentajes de uso de los tres *Puntos de Escape* más comunes.

El cálculo de las *Probabilidades de Escape* y, consecuentemente, de los *Puntos de Escape* en la política peLIFO original es un proceso complejo y costoso en recursos hardware que implica, para cada valor del epCounter, operaciones de redondeo, logaritmos y otros cálculos aritméticos. Además, como hemos visto, hay que almacenar los valores actuales del epCounter y las *Probabilidades de Escape* de la época anterior (en un vector al que denominaremos lastEpCounter). Esto se traduce en dos vectores de $A * s$ elementos, donde A es igual a la asociatividad de la cache, y s es la anchura de palabra requerida para almacenar el valor de cada contador. Tampoco podemos olvidar que es necesario almacenar el dato de la última posición en la que cada bloque produjo

un acierto (lastHitPosition). El coste de almacenar esta información es muy elevado, y tiene la forma de $C * A * \log(A)$, donde C es el número de conjuntos de la cache y A la asociatividad de la misma. Además, la complejidad del control de este sistema es muy elevada y costosa.

Por tanto, y a la luz de la información anterior, es evidente pensar que una importante simplificación del algoritmo que no conllevaría un deterioro excesivo en el rendimiento global consiste en reemplazar el cálculo de los *Puntos de Escape* al inicio de cada fase por el uso de tres *Puntos de Escape* fijos para toda la ejecución de la aplicación, tratando de escoger los tres más utilizados en la política peLIFO original. Eliminar este cálculo nos permite prescindir del epCounter, del lastEpCounter y del costoso lastHitPosition, así como del hardware necesario para efectuar las operaciones aritméticas de cálculo de los *Puntos de Escape*. Además nos permite simplificar el mecanismo de control.

Una primera aproximación consiste en calcular la media de uso de cada *Punto de Escape* a través de las distintas aplicaciones a ejecutar, para poder establecer tres *Puntos de Escape* fijos comunes a todas las aplicaciones. Sin embargo, descartamos esta alternativa dado que valores que funcionan correctamente para algunos benchmarks ofrecen pésimos resultados para otros. Por ello, en la implementación que finalmente llevamos a cabo efectuamos un profiling de cada una de las aplicaciones para de este modo decidir, en cada caso, cuáles son los tres *Puntos de Escape* que seleccionaremos para las sucesivas ejecuciones de dicha aplicación. Los tres valores que escogemos serán aquellas posiciones de la *Fill Stack* en las que mayor número de veces se hayan efectuado reemplazamientos.

Al fijar los tres *Puntos de Escape* podemos hacer una simplificación adicional del mecanismo, consistente en eliminar las épocas LRU que se efectúan al

principio de cada fase (figura 2.5 en página 24). Como se expuso anteriormente, la finalidad de esas épocas es la de servir de entrenamiento para el cálculo de los nuevos *Puntos de Escape*, permitiendo que la cache se llene con nuevos datos y liberando así los sets de bloques pertenecientes a fases anteriores, que puedan contaminar los cálculos de los nuevos *Puntos de Escape*. Dado que en nuestra propuesta dichos valores son fijos, nuestro algoritmo realiza una primera época en modo LRU para permitir que la cache se vaya llenando, y de ahí en adelante se pasa a utilizar épocas peLIFO hasta el final de la ejecución (figura 3.1). Por otra parte, al principio de cada época LRU del algoritmo original se reinician los contadores de Set Sampling. Para mantener un comportamiento equivalente tras eliminar las épocas LRU, extraemos del profiling la duración media de las fases, y en función de ese valor reiniciamos el Set Sampling periódicamente.

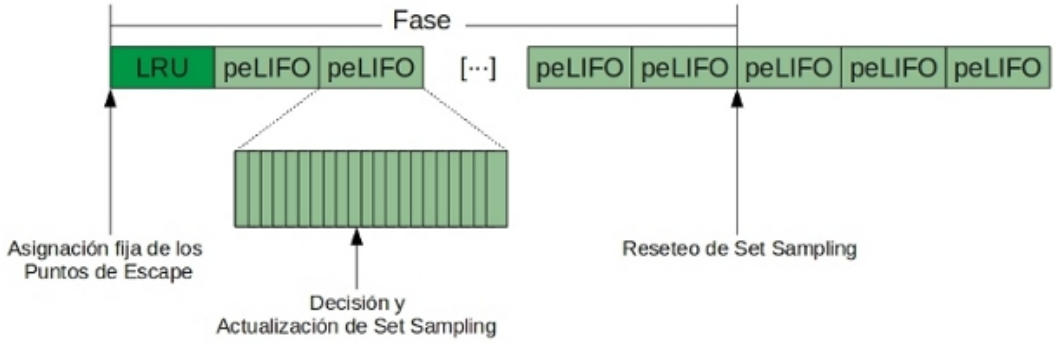

Figura 3.1: Representación esquemática de la política peLIFO-fixPE

3.2. Eliminación de set sampling mediante decisión de grano grueso

Como hemos explicado con anterioridad, la política peLIFO original decide cuál de las cuatro estrategias utilizar (LRU o peLIFO con alguno de los tres *Puntos de Escape*) mediante Set Sampling. Para ello, además del hardware que

hemos visto en la sección 3.1, peLIFO reserva un porcentaje de los conjuntos de la cache (en torno al 6 % de ellos) para probar las distintas políticas. Adicionalmente, emplea seis contadores saturados y una serie de comparadores para decidir la política con mejor rendimiento así como un mecanismo de control dedicado a gestionarlos. La propuesta mostrada en el apartado anterior nos permitía eliminar los recursos destinados al cálculo de los *Puntos de Escape*, pero mantenía intacto el mecanismo de Set Sampling para poder decidir el *Punto de Escape* a usar. Con la intención de reducir todavía más los recursos hardware empleados por la política, planteamos sustituir el Set Sampling por un mecanismo de decisión más simple.

El algoritmo original decide la política a usar en cada reemplazamiento. Sin embargo, consideramos que no es necesario usar un grano tan fino de selección, con el sobrecoste hardware que conlleva. Hemos observado que en gran parte de las aplicaciones evaluadas el *Punto de Escape* elegido en cada reemplazamiento se mantiene constante durante periodos largos de tiempo, en ocasiones de hasta varias épocas. En otros casos, si no constante, observamos cómo un *Punto de Escape* tiene un uso mayoritario con respecto a los demás (figura 3.2: fragmento del benchmark 171.swim). Por tanto es factible escoger un *Punto de Escape* en un momento dado y mantener la misma elección durante periodos largos de duración homogénea, revisando periódicamente nuestra decisión.

En la política original, mostrada en el apartado 2.3, al principio de cada fase se realiza una época LRU durante la que se llena la cache. Al finalizar ésta, se calculan los tres *Puntos de Escape* que se usarán durante dicha fase. A continuación, durante las épocas peLIFO, cada acceso actualiza el mecanismo de Set Sampling, cuyo valores se emplean para decidir el *Punto de Escape* a usar en cada reemplazo. Nuestra propuesta queda reflejada en la figura 3.3. Al principio de cada fase llevamos a cabo una época de entrenamiento para cada

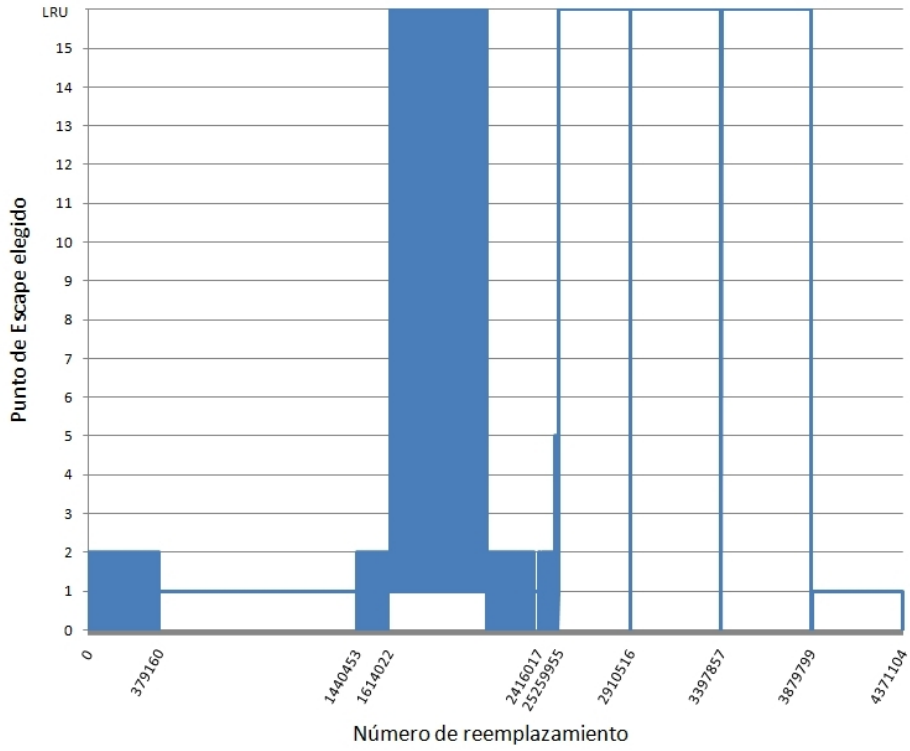

Figura 3.2: Estabilidad de los *Puntos de Escape* a lo largo del tiempo

uno de los posibles *Puntos de Escape* a usar (que incluso pueden ser tantos como el número de posiciones de la *Fill Stack*). En cada una de estas épocas ejecutamos la política peLIFO con el *Punto de Escape* correspondiente, y evaluamos su rendimiento como el número de aciertos en función del número total de accesos[1]. Una vez se ha llevado a cabo el entrenamiento comprobaremos cual ha sido el *Punto de Escape* que ha mostrado un mejor rendimiento, y lo estableceremos como elección constante durante el resto de la fase.

Con esta aproximación eliminamos por completo el mecanismo de Set Sampling. Esto nos permite destinar todos los conjuntos de la cache a la política en uso, eliminando los contadores saturados y los comparadores, y simplificando

[1]Para permitir que en cada época de entrenamiento el conjunto de datos residente en la cache se adapte al *Punto de Escape* evaluado, las estadísticas de accesos y aciertos se considerarán únicamente en la parte final de cada época.

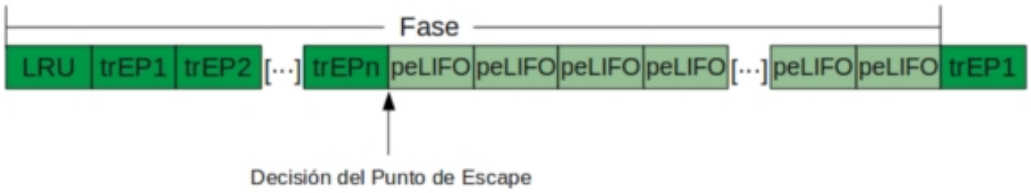

Figura 3.3: Representación esquemática de la política peLIFO-CG

la lógica de control. Al igual que en la propuesta del apartado 3.1, podemos descartar también el hardware destinado al cálculo de los *Puntos de Escape* a usar en cada fase. Sin embargo, mientras que en el apartado anterior lo hacíamos porque asignábamos valores fijos a los *Puntos de Escape* a usar, en este caso se debe a que escogeremos el *Punto de Escape* a emplear de entre los distintos valores para los que llevamos a cabo el entrenamiento. A cambio de todas estas simplificaciones, necesitaremos añadir contadores en los que almacenar la información con la que evaluar el rendimiento de cada *Punto de Escape* durante las épocas de entrenamiento. En nuestra implementación hemos reducido el número de contadores a usar a dos: durante un entrenamiento solo necesitaremos mantener los resultados para el mejor *Punto de Escape* hasta el momento y los que se van obteniendo para el *Punto de Escape* que en esa época se está empleando.

Al igual que en el apartado 3.1, la necesidad de mantener los arrays con la *Probabilidad de Escape* desaparece. Sin embargo, esto provoca que ya no dispongamos de la información necesaria para detectar cambios de fase de la forma en que se lleva a cabo en la política peLIFO original. Dado que en este caso no disponemos de valores de profiling como en la propuesta previa, en esta política no podemos estimar la duración media de una fase. En su lugar, fijamos un valor estático que es potencia de dos para el número de épocas durante las que se aplica la decisión del entrenamiento. Una vez finalizadas esas épocas, se vuelve a realizar una evaluación del *Punto de Escape* más adecuado.

Capítulo 4

Entorno de simulación

Para evaluar la calidad de las políticas propuestas en este proyecto hemos empleado una infraestructura de simulación de cache basada en instrumentación de código junto a un simulador arquitectónico ampliamente usado en el ámbito de la investigación. En este capítulo detallamos las características de cada uno de ellos y explicamos los pormenores de nuestra plataforma de simulación.

4.1. Instrumentación de código con Pin

Pin es una herramienta desarrollada por Intel Corporation y la Universidad de Colorado que permite realizar instrumentación binaria de código [27]. La instrumentación es una técnica consistente en introducir nuevas instrucciones de manera dinámica en un programa en ejecución. De este modo, podemos utilizar información que no está accesible en tiempo de compilación, como la dirección de memoria sobre la que se realizan lecturas y escrituras, el contenido de los registros y la memoria o el orden real en que se planifican y lanzan las

instrucciones en un procesador con ejecución fuera de orden.

Conceptualmente, instrumentar un código tiene dos componentes: un mecanismo que decide dónde se inserta el código y el código a ejecutar en los puntos de inserción. Esos dos elementos se integran en las denominadas Pintools. Una Pintool puede ser concebida como un plugin en el que se especifica el código a ejecutar y los puntos donde éste se insertará. Todo el proceso se lleva a cabo a través de una serie de funciones que hacen de interfaz entre Pin y el código a insertar.

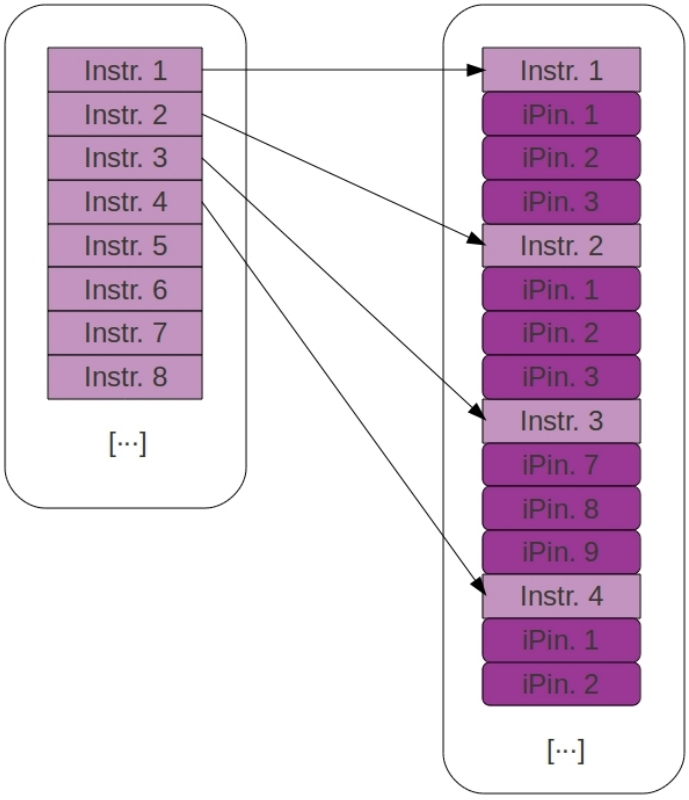

Figura 4.1: Transformación de código para realizar instrumentación

La figura 4.1 ejemplifica el funcionamiento del mecanismo de instrumentación. A la izquierda se representa un flujo de programa normal, en el que las instrucciones se van ejecutando secuencialmente. El código instrumentado

queda representado en la columna de la derecha. Tras cada una de las instrucciones binarias del programa original (en color claro etiquetadas como instr. n), Pin inserta sus propias instrucciones (en oscuro etiquetadas como iPin. k), que podrán recurrir a información acerca de la instrucción asociada a ellas para llevar a cabo sus cómputos.

La instrumentación de código, por tanto, supone un incremento considerable en el tiempo de ejecución de una aplicación. No sólo es necesario que Pin ejecute el código de instrumentación de la Pintool, sino que debe realizar un análisis del programa a instrumentar para decidir en qué punto insertará sus instrucciones. Los datos proporcionados por los desarrolladores indican que, ejecutando únicamente Pin con una Pintool que no añada ninguna instrucción, la sobrecarga en número de instrucciones que sufre el programa a instrumentar ronda el 8 %.

La gran ventaja de este paradigma reside en que es posible instrumentar cualquier aplicación disponiendo únicamente de su binario ejecutable, sin que sea necesario conocer detalle alguno sobre su implementación. El código que se inserta dinámicamente se programa en C o C++, por lo que es posible emplear código ya diseñado y verificado con anterioridad, o recurrir a herramientas escritas en ese lenguaje sin tener que realizar modificaciones en su código.

4.2. Simulador arquitectónico SESC

Originalmente desarrollado por Jose Renau como su proyecto de doctorado, SESC [28] es un simulador arquitectónico cycle-accurate de código abierto ampliamente usado en investigación. El simulador permite explorar una gran variedad de arquitecturas de computadores, desde pipelines sencillos hasta

clusters.

SESC está escrito en C++ y modela, mediante una estructura jerárquica de clases, los diferentes componentes que integran un microprocesador. Al tratarse de un simulador que analiza el comportamiento de computadores completos con precisión de ciclo, ejecutar el programa completo conllevaría un tiempo de simulación que nos llevaría a la obtención de una gran cantidad de información que, realmente, no es útil para los objetivos que perseguimos. Por ello, para este proyecto hemos utilizado exclusivamente la parte del código que simula el funcionamiento de la memoria cache.

La instrumentación desde Pin invoca, con cada instrucción de lectura o escritura, la función correspondiente de SESC que, mediante los argumentos adecuados (identificador de proceso, dirección de acceso y contador de programa), se encarga de realizar la correspondiente actualización en la estructura de cache que estamos simulando. El siguiente extracto de código muestra las funciones de llamada para la lectura de una posición determinada de memoria. Pin inyecta el código que aparece en la función Instruction() después de cada instrucción, de modo que si ésta es de lectura, se invocará a la función readLine() del modelo de cache de SESC.

```
LOCALFUN VOID MemRead(unsigned int tid, ADDRINT addr, ADDRINT pc){
  cacheToSim->mappedSHMem->lockSHM();
  numOpsEjecucion++;
  numReadOpsEjecucion++;
  cacheToSim->readLine(tid,pc,addr);
  cacheToSim->mappedSHMem->unlockSHM();
}
```

```
LOCALFUN VOID Instruction(INS ins, VOID *v){
[...]
  if (INS_IsMemoryRead(ins)){
    numReadOpsCodigo++;
    INS_InsertPredicatedCall(
      ins, IPOINT_BEFORE, (AFUNPTR) MemRead,
      IARG_UINT32, procId,
      IARG_MEMORYREAD_EA,
      IARG_INST_PTR,
      IARG_END);
  }
[...]
}
```

4.3. Simulador multi-core con memoria compartida

El modelo de cache de la que partimos en este proyecto permite la simulación de un conjunto arbitrario de caches de primer nivel con coherencia basada en el protocolo MESI. En este modelo, sin embargo, solo una única instancia de Pin puede acceder a esa estructura, realizando el acceso a las distintas caches mediante threads de ejecución diferentes dentro del mismo proceso (figura 4.2, izquierda). Nuestro interés, sin embargo, reside en simular una cache de dos niveles para un entorno multi-core, donde cada uno de los núcleos cuente con un nivel L1 privado de cache donde ejecutar aplicaciones single-threaded. El segundo nivel de cache será compartido por todos los procesadores (figura 4.2,

derecha).

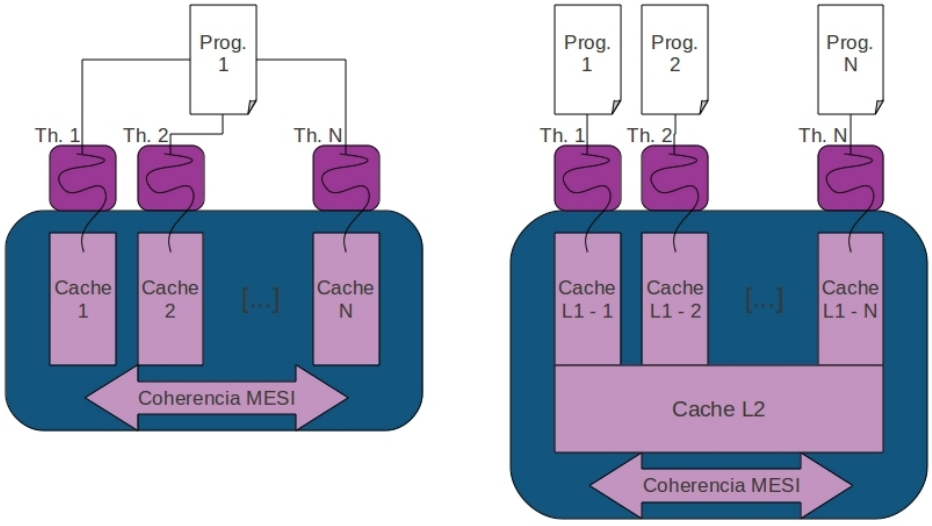

Figura 4.2: Estructura de partida (izq.) y la deseada (dcha.)

Para poder hacer una simulación del modelo que deseamos ha sido necesario realizar varias modificaciones en el modelo del simulador. En primer lugar se generalizó la estructura de la memoria para permitir un número arbitrario de niveles de cache, y se adaptaron las rutinas de búsqueda de bloques y actualización de los mismos para que en caso de fallo de lectura en un nivel k dado, se lance la búsqueda también en el nivel k+1, y en caso de fallo de escritura en dicho nivel k, la escritura se propague desde k hasta J, con J el número total de niveles de la jerarquía de cache.

Para permitir el acceso a la misma estructura de cache por parte de los distintos procesos single-threaded independientes empleamos el paradigma de memoria compartida para hacer que aplicaciones independientes tengan acceso a los mismos datos. Para ello recurrimos a la librería de sistema <sys/shm.h> Ante la excesiva complejidad de adaptar toda la estructura de clases del modelo de cache de SESC para que se instancien los objetos en memoria compartida

[29], hemos desarrollado una implementación en la que los objetos se crean de manera privada mientras que sus atributos, siempre que sean tipos básicos, se mapean en memoria compartida (figura 4.3) mediante el uso de una clase auxiliar diseñada con el objetivo de manejar el proceso de reserva, asignación y liberación de los segmentos de memoria compartida.

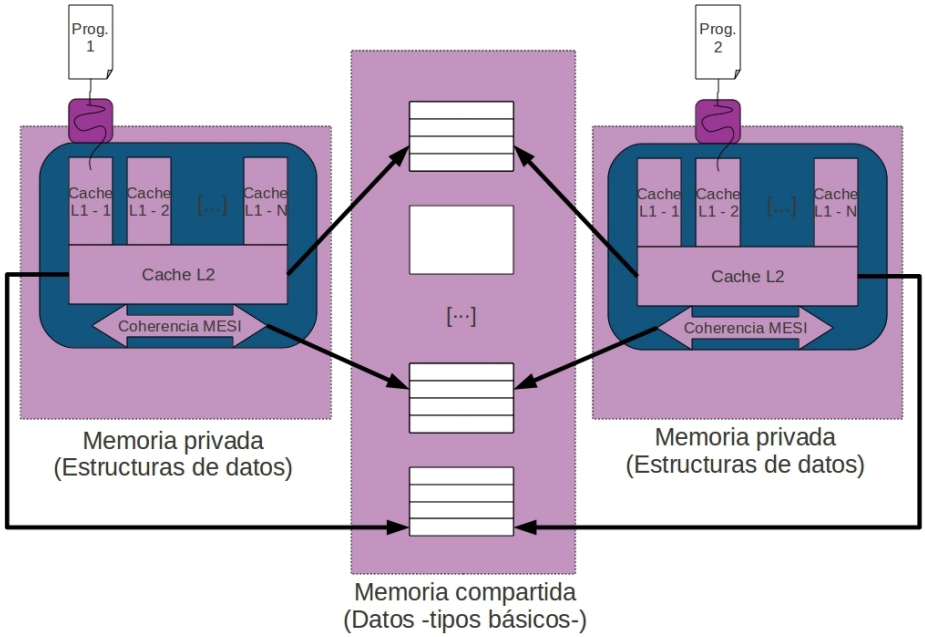

Figura 4.3: Estructura de cache sobre memoria compartida

Para que los resultados obtenidos en este entorno sean correctos cuando el número de procesadores simulados sean dos o más, y por tanto haya niveles compartidos de cache, debemos asegurarnos que no pueda darse la problemática de que dos o más procesos distintos accedan y modifiquen a la vez una misma posición de memoria. De ocurrir eso, los valores que se graban en memoria podrían ser erroneos. Para evitar esto, hemos dotado al simulador de mecanismos de exclusión mutua que restringen el acceso a la sección crítica a un único proceso cada vez. El problema de hacer esto es que el tiempo de simulación se incrementa enormemente, haciendo que la ejecución de cuatro

benchmarks sobre nuestro entorno se prolongue durante varios días.

Tratando de obtener mediciones representativas sin que el tiempo de ejecución sea tan alto, nuestro entorno de simulación integra también SimPoint [30]. Se trata de una aplicación que realiza un análisis de los pares programa/entrada, identifica las distintas partes del programa donde el comportamiento es similar, y escoge una de cada tipo como representativo del comportamiento del resto. Al lanzar el programa, se ejecutarán únicamente los fragmentos representativos identificados, mientras que los resultados del resto únicamente se estimarán basándose en la simulación exhaustiva de los primeros. Gracias a este sistema podemos rebajar los tiempos de ejecución de varios días a aproximadamente diez o doce horas.

4.4. Benchmarks y configuraciones

Para validar los resultados de nuestras propuestas hemos utilizado las suites de benchmarks SPEC CPU 2000 y SPEC CPU 2006, y hemos lanzado las ejecuciones sobre un cluster en el que cada uno de sus nodos cuenta con dos Dual Core AMD Opteron Processor 270 que comparten una memoria de 4GB. Hemos ejecutado dos tipos distintos de pruebas.

En primer lugar, probamos nuestras propuestas para un entorno single-core con dos niveles de cache cuya configuración se indica en la tabla 4.1. Dada la gran velocidad de la infraestructura de simulación que utilizamos, los benchmarks fueron ejecutados hasta el final, empleando para el profiling las entradas de entrenemiento, y para producción las de referencia.

Posteriormente, probamos esas mismas propuestas en un sistema multi-core, en el que se simulaban cuatro procesadores en ejecución, cada uno de ellos

	Single-core	**Multi-core**
Tamaño Cache L1	32 KB	32KB
Asociatividad Cache L1	4 vías	4 vías
Tamaño de Bloque Cache L1	32 B	32 B
Tamaño Cache L2	2 MB	8 MB
Asociatividad Cache L2	16 vías	16 vías
Tamaño de Bloque Cache L1	512 B	512 B

Tabla 4.1: Configuración de la cache simulada.

Identificador de mezcla	**Benchmarks componentes**
MIX01	183, 429, 462, 482
MIX02	181, 183, 433, 482
MIX03	181, 183, 429, 482
MIX04	181, 183, 470, 482
MIX05	172, 403, 429, 462
MIX06	172, 433, 470, 482
MIX07	181, 183, 429, 462

Tabla 4.2: Mezclas para la evaluación en multi-core.

con una ejecución independiente, con un primer nivel de cache L1 privado y un segundo nivel L2 compartido entre todos ellos. La configuración de esta cache está recogida en la tabla 4.1, mientras que en la tabla 4.2 se recogen las distintas mezclas de programas de entrada que se emplearon. En este caso no se lanzaron los benchmarks hasta el final, sino que se hizo uso de SimPoint, ejecutando únicamente secciones significativas del código de las cuatro aplicaciones en paralelo. En este caso se utilizaron las entradas de referencia tanto para el profiling como para las ejecuciones de producción.

Para probar la política presentada en la sección 3.2, debemos escoger un

valor fijo de épocas peLIFO durante las que mantener la decisión del entrenamiento, además de decidir sobre qué *Puntos de Escape* efectuaremos el mismo. Para nuestras pruebas llevaremos a cabo entrenamiento sobre los 16 *Puntos de Escape*, y mantendremos la decisión constante durante 128 épocas, destinando así menos de un 10 % del tiempo de ejecución total a entrenamiento.

Capítulo 5

Resultados experimentales

En este capítulo mostramos los resultados que hemos obtenido en los experimentos que hemos realizado con nuestras propuestas. Hemos puesto a prueba estas políticas en dos ámbitos distintos. Por una parte, en un entorno single-core, para el cual la política peLIFO está originalmente concebida, y posteriormente empleamos dicha metodología en un sistema multi-core con las características descritas en el apartado anterior.

5.1. Entorno single-core

Para comprobar los resultados de nuestras propuestas tomamos como referencias las políticas LRU y Probabilistic Escape LIFO original (peLIFO) y comparamos los resultados obtenidos por ellas con las de la política que emplea los tres *Puntos de Escape* fijos obtenidos por profiling (peLIFO-fixEP) mostrada en el apartado 3.1, así como con la consistente en eliminar el Set Sampling mediante decisión de grano grueso y entrenamiento (peLIFO-CG) del apartado 3.2.

La figura 5.1 muestra las tasas de fallos de cada una de las políticas para los doce benchmarks que hemos considerado, y la tabla 5.1 la media de porcentajes de fallos de cache para ellas. En la tabla 5.1 vemos cómo los resultados de la política peLIFO original logran reducir sensiblemente la tasa de fallos con respecto a una estrategia LRU. Mientras que la tasa de fallos para LRU asciende hasta el 25.23 % de fallos de media, peLIFO se queda únicamente en el 23.51 %. Tomando en consideración estos valores, observamos el rendimiento de nuestras dos propuestas.

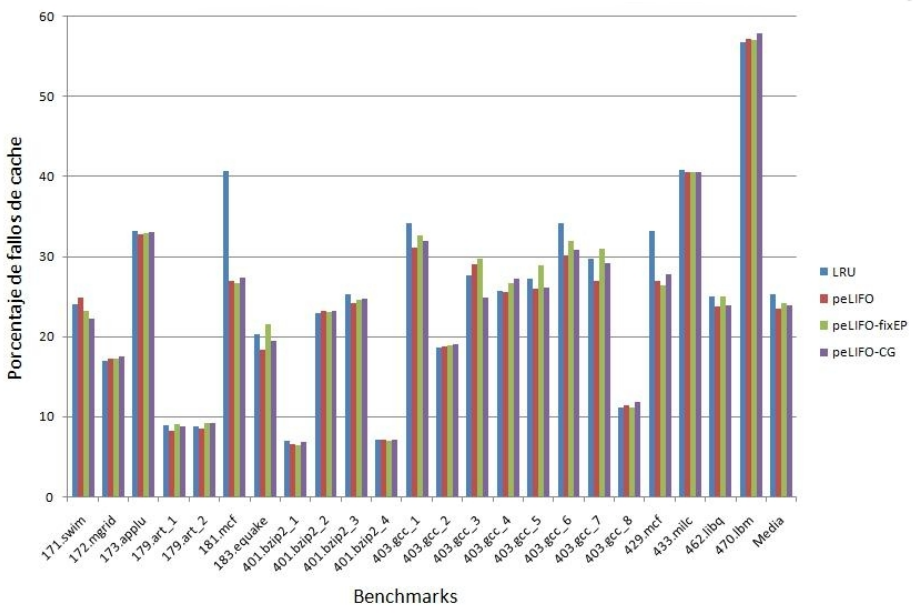

Figura 5.1: Tasas de fallos para los distintos benchmarks single-core

En el capítulo 3 ya apuntamos la ganancia en cuanto a overhead hardware que las políticas allí expuestas obtenían. En la figura 5.1 y la tabla 5.1 constatamos que tanto peLIFO-fixEP como peLIFO-CG logran el objetivo de únicamente sufrir un deterioro suave de sus resultados. Vemos así que la política peLIFO-fixEP provoca un 0.71 % más de fallos que peLIFO. Por su parte, vemos que la tasa de fallos de peLIFO-CG es solo un 0.39 % peor que la pro-

Política	**Media de fallos**
LRU	25,23 %
peLIFO	23,51 %
peLIFO-fixEP-óptimo	23,83 %
peLIFO-fixEP	24,22 %
peLIFO-CG	23,90 %

Tabla 5.1: Media de fallos en single-core

puesta original.

Aunque puede que los porcentajes de reducción de fallos con respecto a la política LRU no parezcan muy significativos, es necesario resaltar que nos encontramos en el último nivel de la cache, y por tanto, un fallo en este nivel produce un acceso a la memoria principal, con la penalización que esto supone. Atendiendo a la información proporcionada en [9], mientras que un acierto en la cache L2 tiene una latencia de 9 ciclos, un acceso a la memoria principal tarda un total de 100ns, lo que en un procesador a 4GHz como el de referencia supone una penalización de 400 ciclos de reloj. Una reducción en la tasa de fallos como la obtenida por estas políticas supone, por tanto, una gran mejora en el rendimiento global del sistema.

Al realizar las mediciones para la política peLIFO-fixEP pudimos observar que la elección de los *Puntos de Escape* fijos que efectuemos puede tener un impacto muy importante en el rendimiento de la política de reemplazamiento. Es por ello que nos interesa saber si las entradas de entrenamiento utilizadas para decidir esos valores son lo suficientemente representativas del comportamiento del programa con las entradas de referencia. Para hacer esta valoración suponemos como *Puntos de Escape* óptimos las tres posiciones de la *Fill Stack* en las que se efectuaron más reemplazamientos durante la ejecución

de la aplicación con entradas de referencia. Dichos valores aparecen en la tabla 5.1 identificados como peLIFO-fixEP-óptimo. Comprobamos así que la tasa de fallos es notablemente más baja que la obtenida con los *Puntos de Escape* deducidos de la ejecución con entradas de entrenamiento. Podemos afirmar, por tanto, que el perfil de transacciones de memoria de dichas entradas no es suficientemente representativo del comportamiento de los conjuntos de referencia, y que unos benchmarks que produjesen unos *Puntos de Escape* más próximos a los óptimos conducirían efectivamente a una tasa de fallos más baja. En la tabla 5.2 vemos tres ejemplos de los *Puntos de Escape* empleados en los casos de las entradas de referencia (ref-EP) frente a los obtenidos con entradas de entrenamiento (train-EPs). Como podemos ver, hay casos como en 403.gcc donde varios de los *Puntos de Escape* coinciden, mientras que en otros como 401.bzip2 tienen poco o nada que ver.

Benchmark	**ref-EPs**	**train-EPs**
183.equake	1, 2, 15	1, 14, 15
401.bzip2	2, 4, 5	6, 10, 13
403.gcc	0, 1, 5	0, 1, 15

Tabla 5.2: *Puntos de Escape* con entradas de entrenamiento y referencia

En el caso de la política peLIFO-CG, en cambio, no sufrimos este efecto. Aunque las épocas de entrenamiento pueden limitarse a un subconjunto de los posibles *Puntos de Escape* (en caso, por ejemplo, de que el número de estos fuese excesivamente grande), hemos realizado nuestras pruebas considerándolos todos, y por tanto, siempre podremos emplear al más prometedor. Gracias a este efecto es posible incluso, como mostraremos más adelante, que los *Puntos de Escape* que escojamos resulten incluso más adecuados que los que la propia política peLIFO utilizaría, logrando en ocasiones mejoras con respec-

to al planteamiento original. Los resultados experimentales demuestran que el rendimiento medio de esta propuesta está próximo a la de peLIFO-fixEP-óptimo, con una mejora significativa con respecto a peLIFO-fixEP. Recordamos, además, que es con esta propuesta con la que logramos una mayor reducción de la cantidad de hardware usado para implementar la política, maximizando así la relación entre coste y rendimiento.

Resulta también interesante comprobar la distribución de los *Puntos de Escape* seleccionados por cada aplicación para las distintas políticas. Los resultados para los tres benchmarks que presentamos aquí muestran tres situaciones distintas con las que nos podemos encontrar al aplicar las técnicas presentadas en este trabajo. En primer lugar, observamos la distribución de la selección de los *Puntos de Escape* para el benchmark 401.bzip2_2 (figura 5.2), en el que apenas existe diferencia en la tasa de fallos entre las políticas peLIFO original y nuestras dos propuestas.

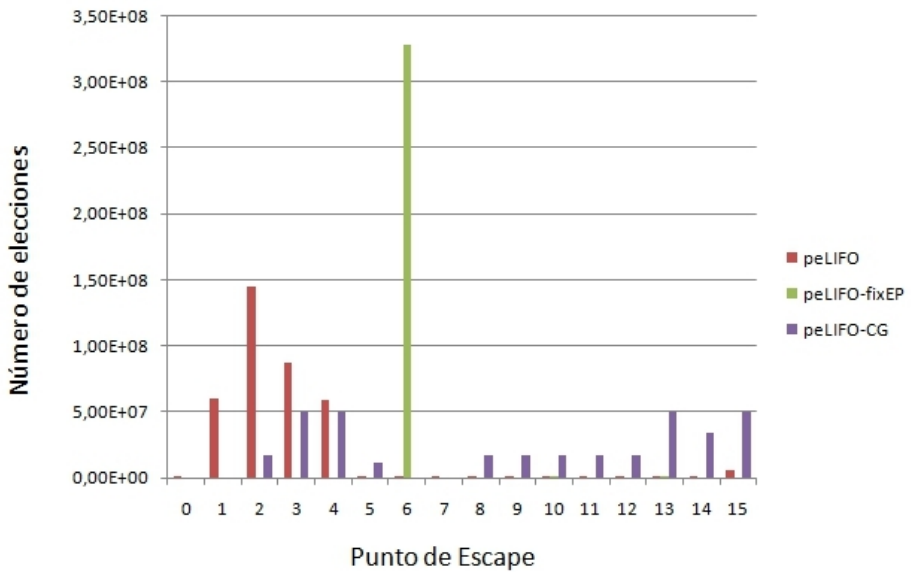

Figura 5.2: *Puntos de Escape* elegidos en 401.bzip2_2

Vemos aquí cómo la política peLIFO-fixEP escoge el *Punto de Escape* más

cercano a la cabeza de la *Fill Stack* con más frecuencia de entre los tres obtenidos mediante profiling (tabla 5.2), mientras que la política peLIFO-CG, a pesar de escoger un porcentaje considerable de veces sus *Puntos de Escape* en posiciones semejantes a las de la política original, estima que el punto óptimo se encuentra en más ocasiones cerca de la cola de la *Fill Stack*.

Existe la posibilidad de que, por las características del programa a ejecutar, las simplificaciones propuestas en este trabajo empeoren considerablemente los resultados. El caso más claro de esta situación es el de 403.gcc_7, donde la diferencia en las tasas de fallos es de hasta el 4 % (figura 5.3).

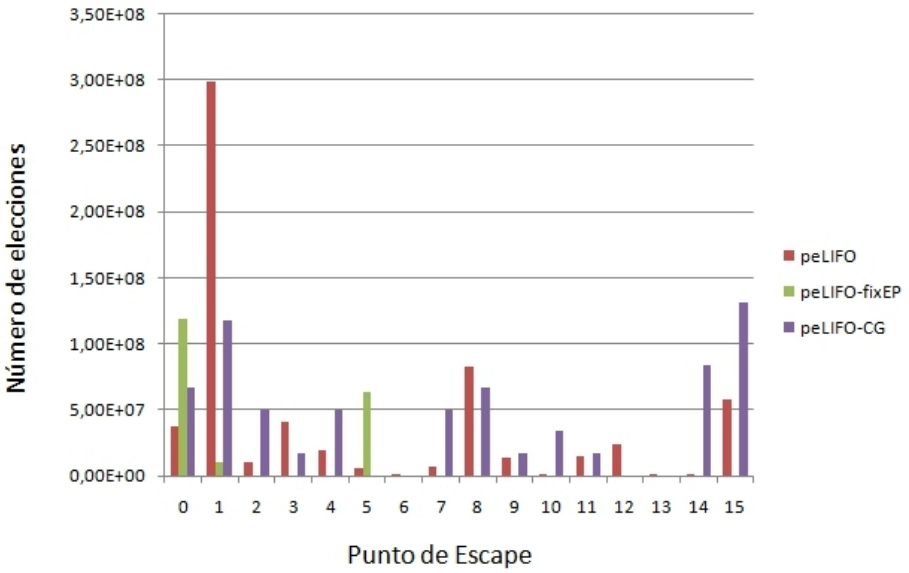

Figura 5.3: *Puntos de Escape* elegidos en 403.gcc_7

Las decisiones de los *Puntos de Escape* de la política peLIFO muestran que, aunque es evidente una fuerte preferencia por el *Punto de Escape* 1, existe una diversidad de elecciones notable. La influencia de este hecho en la política peLIFO-fixEP es importante. Al fijar tres *Puntos de Escape*, estamos perdiendo una amplia cantidad de información acerca del comportamiento en el resto de posiciones de la *Fill Stack*. Esto ocasiona, en caso de que los *Puntos de*

Escape que se escogen en la política original sean muy variados, que los valores que nuestra propuesta elige estén lejos del óptimo. Por otra parte, si nos fijamos en la cantidad de elecciones que se efectúan en la política peLIFO y en peLIFO-fixEP, se constata rápidamente que el número de reemplazamientos en la primera política es mucho mayor que en el de nuestra propuesta. Si los *Puntos de Escape* fijos no ofrecen buenos resultados, el mecanismo de Set Sampling acabará inclinándose por emplear la poltica LRU, cuyas elecciones no están contempladas en las figuras aquí mostradas. A ello se debe, por tanto, la amplia diferencia en el número de elecciones de una política a otra.

Tanta variedad de elecciones en los *Puntos de Escape* también tiene efectos nocivos para el rendimiento de la poltica peLIFO-CG. En este caso, el deterioro de las tasas de aciertos puede suceder en caso de que no haya ningún *Punto de Escape* que se mantenga como elección mayoritaria durante una larga cantidad de reemplazamientos, provocando por tanto que el Punto que escogemos no sea el adecuado durante periodos importantes del tiempo de ejecucion, incrementando consecuentemente la tasa de fallos.

Finalmente, también encontramos casos en los que la aplicación de nuestras propuestas conlleva mejoras de rendimiento. Tenemos así el caso de 171.swim, cuyas elecciones de *Puntos de Escape* mostramos en la figura 5.4.

En este caso, la mejora de rendimiento se puede explicar por el hecho de que la política peLIFO escoge sus *Puntos de Escape* basándose, como vimos, en los tres primeros cambios bruscos de la *Probabilidad de Escape* en la *Fill Stack*. Sin embargo, cabe la posibilidad de que las posiciones más adecuadas para realizar reemplazamientos en la *Fill Stack* sean valores posteriores, en cuyo caso la política peLIFO-CG puede identificar más acertadamente dichas situaciones. En este caso concreto, observamos un importante descenso de la

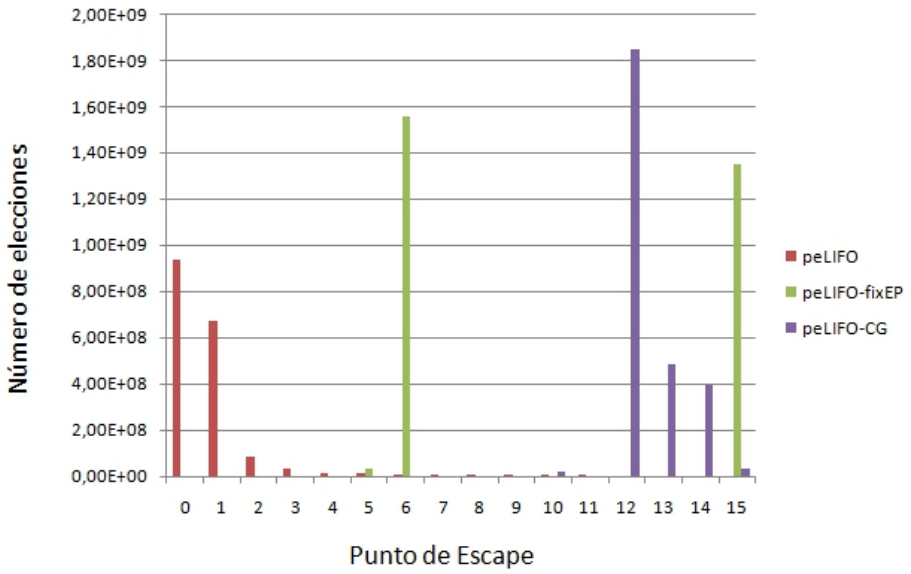

Figura 5.4: *Puntos de Escape* elegidos en 171.swim

tasa de fallos desde un 24.81 % de la política peLIFO (empeorando incluso los resultados obtenidos por la política LRU) hasta un 22.2 % en el caso de la simplificación aquí presentada.

5.2. Entorno multi-core

Dado que la política peLIFO original obtenía también ganancias muy notables con respecto a sus competidoras para entornos multi-core, también nosotros realizamos la evaluación de nuestras propuestas en este casos. Sin embargo, debido a las características de nuestro entorno de simulación, estas pruebas cuentan con ciertas limitaciones que es necesario resaltar.

Como hemos señalado en el capítulo 4, el control de acceso a las estructuras de cache en memoria compartida se lleva a cabo mediante un mecanismo de exclusión mutua basada en mutex. Esto provoca que el orden en que se en-

trelazan finalmente los distintos accesos posee un alto grado de aleatoriedad, basándose en qué proceso se haga en primer lugar con el control del mutex una vez éste se libera. Aunque simula eficazmente el comportamiento real de un procesador multi-core, no podemos tener la seguridad de que el orden de llamadas y reemplazamientos sea el mismo, por lo que los resultados obtenidos no reflejan el comportamiento de las políticas de reemplazamiento bajo las mismas cadenas exactas de reemplazamientos. Hemos podido comprobar que existe cierto grado de variación en los resultados obtenidos en distintas ejecuciones para un mismo par benchmark/política, por lo que los resultados mostrados aquí para cada uno de ellos es la media entre de dos ejecuciones distintas, con el fin de obtener valores promedio.

Una vez hecho notar esto, pasamos a presentar los resultados obtenidos en las pruebas para entornos multi-core que hemos llevado a cabo.

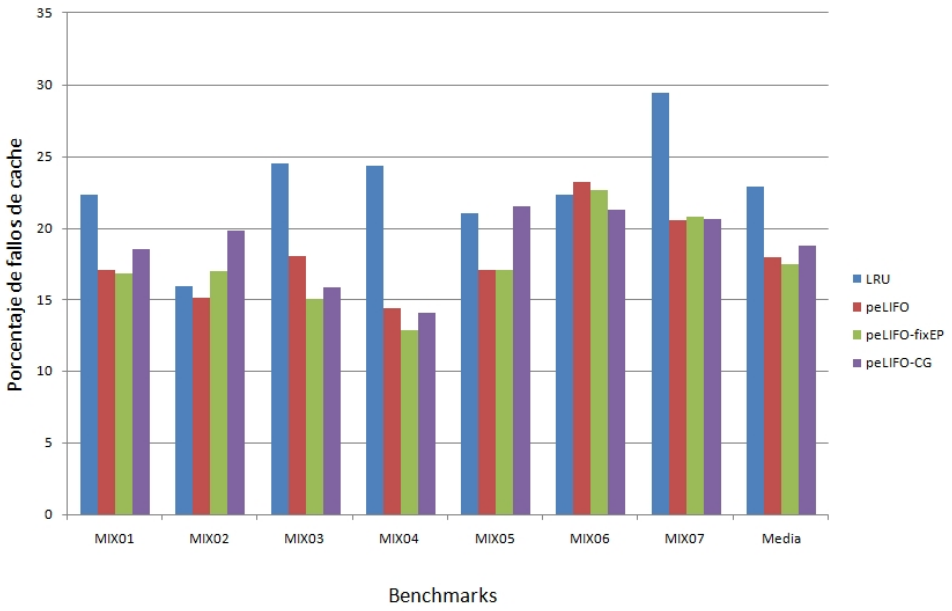

Figura 5.5: Tasas de fallos para los distintos benchmarks multi-core

Comprobamos en este caso cómo, a pesar de la existencia de cierto grado

Política	**Media de fallos**
LRU	22,89 %
peLIFO	17,95 %
peLIFO-fixEP	17,49 %
peLIFO-CG	18,83 %

Tabla 5.3: Media de fallos en multi-core

de variación en los resultados entre ejecuciones, las tasas de fallos de nuestras propuestas, tanto peLIFO-fixEP como peLIFO-CG, se mantienen muy próximas a las de la política original. Esto indica que las simplificaciones de diseño presentadas en este trabajo funcionan también de forma eficiente en entornos multi-core.

Capítulo 6

Conclusiones y trabajo futuro

En este trabajo hemos presentado dos técnicas de simplificación de la política Probabilistic Escape LIFO que buscan maximizar la relación entre la calidad del algoritmo y su coste de implementación hardware. A la luz de los resultados mostrados, podemos concluir que hemos logrado dos implementaciones con un coste sustancialmente menor que el de la política peLIFO original mientras que mantenemos el deterioro del rendimiento con respecto a la propuesta de la que partíamos en tasas poco significativas tanto en un entorno single-core como en multi-core.

Adicionalmente, hemos comprobado que existen ocasiones en las que nuestra política de decisión de grano grueso (peLIFO-CG) realiza una mejor estimación los *Puntos de Escape* sobre los que realizar el reemplazamiento, provocando mejoras en la tasa de aciertos. Respecto a nuestra política basada en profiling (peLIFO-fixEP), hemos observado que las entradas de entrenamiento de los benchmarks no presentan un perfil de transacciones de memoria representativo, lo que nos lleva a cometer un mayor número de fallos de cache. Escoger *Puntos de Escape* más cercanos a los óptimos nos garantiza mejores

resultados.

Existen diversas vías de trabajo futuro abiertas a partir de los resultados obtenidos. En primer lugar, como hemos indicado en la sección 5.2, las mediciones efectuadas para entornos multi-core no se llevan a cabo sobre series idénticas de reemplazamientos. Para poder efectuar pues una evaluación totalmente equitativa, podría adoptarse una solución consistente en obtener, mediante instrumentación de código, una traza de ejecución, y utilizarla posteriormente como entrada para el simulador de cache implementado en SESC.

Por otra parte, la política peLIFO original está planteada para los últimos niveles de cache privada, con una sola aplicación ejecutándose sobre ella. Sin embargo, los resultados experimentales muestran que también se comporta realmente bien en entornos multi-thread y multi-core. Siendo así, existen ya proyectos en marcha que buscan incorporar información thread-aware al mecanismo con el fin de mejorar todavía más su rendimiento. Este tipo de mejoras se están demostrando efectivas, pero el problema de la complejidad hardware no solo se mantiene, sino que existe cierto grado de incremento debido a las nuevas consideraciones que dichos sistemas tienen en cuenta para abarcar la información referente a los hilos de ejecución. Por ello, combinar dichas optimizaciones con las simplificaciones aquí mostradas podría conllevar a mejorar las prestaciones de esas políticas manteniendo un coste hardware muy ajustado.

Finalmente, aunque las simplificaciones propuestas en este trabajo ofrecen una reducción muy considerable del hardware empleado y de su control, seguimos manteniendo la *Fill Stack* como componente necesario para la implementación. Sería posible, sin embargo, abordar la posibilidad de eliminar esta estructura, también costosa, mediante técnicas similares a las empleadas en Tree-LRU, para de este modo disminuir todavía más el sobrecoste hardware.

Publicaciones realizadas

- S. Sepúlveda, E. Sedano, D. Chaver, F. Castro, L. Piñuel, F. Tirado. "*Simplificación y extensión a un entorno multi-core de la política de reemplazamiento Probabilistic Escape LIFO*". CEDI 2010.

Referencias

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2002.

[2] L. Liu, Z. Li, and A. H. Sameh. Analyzing Memory Access Intensity in Parallel Programs on Multicore. *Proc. of the 22nd annual Intl. Conf. on Supercomputing*, pages 359–367, 2008.

[3] G. Z. Chrysos and J. S. Emer. Memory Dependence Prediction Using Store Sets. *Proc. 25th Intl. Symp. on Computer Architecture*, pages 142–153, 1998.

[4] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value Locality and Load Value Prediction. *Proc. 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[5] J. Gonzalez and A. Gonzalez. Speculative Execution via Address Prediction and Data Prefetching. *Proc. of the Intl. Conf. on Supercomputing*, pages 196–203, 1997.

[6] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. *ISCA-27*, pages 128–135, 2000.

[7] T. Chen and J.L. Baer. Effective Hardware-based Data Prefetchting for High Performance Processor. *EEE Transactions on Computers, Vol 44, N 5*, pages 609–623, 1995.

[8] L. A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Systems Journal*, pages 78–101, 1966.

[9] M. Chaudhuri. Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches. *Intl. Symp. on Microarchitecture*, pages 401–412, 2009.

[10] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-aware Cache Replacement. *Int. Symposium on Computer Architecture*, pages 167–177, 2006.

[11] M. K. Qureshi et al. Adaptive Insertion Policies for High Performance Caching. *Proc. of the 34th Intl. Symp. on Computer Architecture*, pages 381–391, 2007.

[12] D. Rolán, B. B. Fraguela, and R. Doallo. Adaptive Line Placement with the Set Balancing Cache. *Proc. of the 42nd Intl. Symp. on Microarchitecture*, pages 529–540, 2009.

[13] N. P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache Prefetch Buffers. *Intl. Symp. on Computer Architecture*, pages 364–373, 1990.

[14] M. K. Qureshi and Y. N. Patt. Utility-based Cache Partitioning: A Low-overhead High-performance Runtime Mechanism to Partition Shared Caches. *Proc. 39th Intl. Symp. on Microarchitecture*, pages 423–432, 2006.

[15] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. *Proc. 36th Intl. Symp. on Computer Architecture*, pages 174–183, 2009.

[16] T. R. Puzak. Analysis of Cache Replacement-algorithms. *Electronic Doctoral Dissertations for UMass Amherst. Paper AAI8509594*, 1985.

[17] J. T. Robinson. Generalized Tree-LRU Replacement. *IBM Research Report RC23332 (W0409-045)*, 2004.

[18] H. Ghasemzadeh, S. Mazrouee, and M. R. Kakoee. Modified Pseudo LRU Replacement Algorithm. *13th Annual IEEE Intl. Symp. and Wrkshp. on Engineering of Computer Based Systems*, 2006.

[19] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. *Proc. of the 29th Intl. Symp. on Computer Architecture*, 2002.

[20] J. B. Sartor, S. Venkiteswaran, K. S. McKinley, and Z. Wang. Cooperative Caching with Keep-Me and Evict-Me. *9th IEEE Annual Wrkshp. on the Interaction between Compilers and Computer Architectures*, 2005.

[21] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the Compiler to Improve Cache Replacement Decisions. *Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2002.

[22] J. Abella, A. Gonzalez, X. Vera, and M. O'Boyle. IATAC: A Smart Predictor to Turn-Off L2 Cache Lines. *ACM Trans. Architecture and Code Optimization*, 2005.

[23] M. Kharbutli and Y. Solihin. Counter-Based Cache Replacement and Bypassing Algorithms. *IEEE Transactions on Computers, vol. 57, no. 4*, pages 433–447, 2008.

[24] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. *Proc. of the 41st Intl. Symp. on Microarchitecture*, pages 222–233, 2008.

[25] A. Jaleel et al. Adaptive Insertion Policies for Managing Shared Caches. *Proc. of the 17th Intl. Conf. on Parallel Architecture and Compilation Techniques*, pages 208–219, 2008.

[26] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors. *Proc. 13th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 135–144, 2008.

[27] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[28] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator. `http://sesc.sourceforge.net`.

[29] L. Hon. Using Objects in Shared Memory for C++ Application. *CASCON '94 Proc. of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 29, 1994.

[30] B. Calder, T. Sherwood, G. Hamerly, and E. Perelman. *Performance Evaluation and Benchmarking*, chapter 7. CRC Press, 2005.